Concurrent Model Checking - Systematic testing/Stateless Model Checking

Niklas Klipphahn
University of Kaiserslautern
n_klipphahn12@cs.uni-kl.de

Abstract

Concurrency is a necessary part in today’s software systems. A concurrent system consists of elements - called processes - that operate concurrently and communicate with each other via shared communication objects. In contrast to strictly sequential programs, concurrent programs are significantly harder to design and even harder to test. Common testing techniques used in the industry simply run test suites which test different sets of representative inputs. This way of testing is not sufficient for concurrent systems. Their non-deterministic and reactive nature results in hard to reproduce errors and unexpected behavior that even surprises experienced programmers. Model checking is a promising alternative that tries to address the problems traditional testing has. Generally, model checkers systematically explore a program’s state space. When trying to explore such a state space, we are confronted with the state explosion problem. The number of possible states grows exponentially with the program’s size because of their non-deterministic nature. This makes standard state space search methods useless, due to limited time, and resources. A better way to approach the problem is to use stateless model checking. This paper aims to introduce several ways to avoid state-space explosion through stateless model checking.

1 Introduction

Most modern computer systems cannot go without concurrent parts. Elementary parts of a concurrent system are processes that operate concurrently and communicate with each other via shared communication objects. Concurrent software systems are hard to design and even harder to test. Traditional testing techniques, which are still used in the industry, are not sufficient. They often use stress-testing, where software is executed under heavy loads in hope of producing errors. Errors that occur during stress-testing often are hard to reproduce and the coverage of such tests is questionable and probably very low in regard to concurrency. Threads or processes of a concurrent software system often have unexpected interleaving which results in unexpected behavior which even experienced programmers are unable to predict. Model checking is a promising alternative to traditional testing. It addresses limitations and problems traditional testing has. A number of different model checkers exist, even for common programming languages such as C++ or Java. A model checker systematically executes the schedule of every process and verifies that desired properties of the program are satisfied. In other words model checkers systematically explore the state space of programs. Standard state space search methods are confronted with the state explosion problem. The number of possible states grows exponentially with the programs size due to their non-deterministic nature.

A better way to approach this problem is stateless model checking, another way to efficiently search the
state space, is stateless model checking. This paper aims to give an overview over concurrent system and stateless model checking, as well as presenting several methods to perform stateless search. The paper is organized as follows. First we give an overview over stateless model checking in Section 3. Then, in Section 2 we present some background definitions to formally describe a concurrent system, so we can work with them easily when we need them to describe the different methods in detail. Several types of Partial order reduction are presented in Section 5 followed by Iterative Context Bounding in Section 4 and Partial Order Reduction for Context-Bounded Exploration in Section 6. Section 7 is a synopsis of related work and Section 8 shows our conclusion.

2 Background Definitions

In this chapter we introduce definitions for modeling of concurrent systems. Concurrent systems consist of processes that can interact with each other and act independently from each other in parallel circuits. We assume that those processes are finite-state, which means the number of states a process can reach is limited. We label a finite set of processes or threads \( \mathcal{P} \). Each process executes a sequence of operations from a deterministic, sequential program written in a language such as C++ or Java. Processes are able to communicate with each other through objects. They perform atomic operations on objects, such as shared variables, locks, FIFO Buffers or semaphores. Even though threads are a particular kind of process that run in a shared memory space, we use the terms process and thread interchangeably.

Operations executed by the processes are called visible, if they are executed on a communication object. Else they are called invisible. Operations are blocking if they currently cannot be completed. This occurs if for example another process currently acquired the lock for an object. Thus, it is impossible for invisible operations to block, because only visible operations, on shared/ communication objects may block. A finite set of communication objects \( \mathcal{O} \) consists of objects that are defined as pair \((V, OP)\). \( V \) is the domain for that object and \( OP \) are the operations that can be executed on that object.

At any given time the system is said to be in a state. The state of a concurrent system is composed of the local state of each process and of the shared state of all communication objects. The global state is defined as state in which every next operation of all processes is visible. Assuming that every process executes visible operations at some time we get a unique first global state which we call initial global state \( s_0 \in \mathcal{S} \).

A transition moves the system from one state to another by executing visible operations of a process. The chosen process may perform a finite sequence of invisible operations, before the next visible operation. The set of all transitions of the system is represented by \( T \). A transition \( t \) is either enabled or disabled. If \( t \) can be executed in a global state \( s \), it is called enabled. Otherwise, it is called disabled and execution is currently not possible, because the visible operation on a shared object is blocking. We assume that the number of invisible operations is finite and therefore enabled transitions always terminate. After executing \( t \) the system reaches a new state \( s' \) which is the successor of \( s \) by \( t \).

Let \( s \xrightarrow{t} s' \) denote the execution of transition \( t \) that leads from global state \( s \) to \( s' \). If a finite set of transitions leads from \( s \) to \( s \) we say that \( s' \) is reachable from \( s \). A state with no enabled transition is called a deadlock or terminating state.
A labeled formal concurrent system is a tuple \((\mathcal{P}, \mathcal{O}, T, s_0)\) where

1. \(\mathcal{P}\) is a finite set of processes
2. \(\mathcal{O}\) is a finite set of objects
3. \(T\) is a finite set of transitions
4. \(s_0\) is the initial global state of the system

In our definitions we combine invisible operations with the last visible one in one transition. Therefore, we ignore the negligible effect of invisible operations in regard to concurrency. We reduce the state space and are still able to detect deadlocks and assertion violations. In another work [4] it is proven that this is sufficient.

3 Stateless Model Checking

To test concurrent software more efficiently the idea of stateless model checking has emerged to combat the problem that state space explosion is. It was first proposed in Verisoft [4]. In contrast to stateful model checking, stateless model checking searches the state space of concurrent software without capturing specific program states. Rather than storing the concrete state of the system, stateless model checking uses abstract states, where each state is uniquely identified by the sequence of transition that lead from the initial state to this state. To achieve this, a special scheduler is used to control the non-determinism of the software which systematically enumerates different total order in which concurrent events of the program can occur.

As a side note, we assume that concurrency is the only source of non-determinism. [14]

Some stateless exploration techniques use an execution tree to control the exploration progress. This execution tree abstractly stores the software’s states. Nodes represent non-deterministic choice-points and edges represent program state transitions. Following a path from the root of the execution tree to a leaf, results in an execution sequence of transitions needed to reach the state represented in the leaf from the state represented in the root. Therefore, enumeration of branches from that tree corresponds to enumeration of different sequences of state transitions. It’s rather easy to keep track of already explored state transitions with the set of explored branches of a partially explored execution tree. It’s then easy to see which sequences of transitions in previous executions were used and generate sequences for future executions that explore new parts of the execution tree.

There are a number of ways to explore the execution tree. A well-known algorithm to do that is depth-first search. For complex systems depth-first search is not fast enough which is why we need tools to reduce the state space or avoid executing the same sequence of program states multiple times. Such tools are for example:

1. Partial Order Reduction
2. Iterative Context Bounding

Of course those tools have their own advantages and disadvantages and specific problems that they can address.
One difficulty when model checking and stateless model checking is that often \textit{Sequential Consistency} is presupposed as execution model. It assumes that each different process has a total order in their execution of instruction. But concurrency may not be the only source of non-determinism. Modern processors implement \textit{weak memory models}. Such a model allows more behaviors than just those of Sequential Consistency. Modern processors don’t write to memory directly. They have to write to a store buffer first, then to a cache and then finally to memory. For example, while such a write action is still in progress a read may occur before the new value is actually available for all processors in memory. This issue can be addressed when using partial orders [1]. We will not further discuss implementations that use a weak memory model in detail. For simplicity reasons we presuppose the Sequential Consistency model.

\textbf{Figure 1: Dining Philosophers Livelock (from [11])}

\textbf{Shared Communication Variables:}
Object \texttt{fork1,fork2;}

\begin{tabular}{ll}
\textbf{Thread1:} & \textbf{Thread2:} \\
1 \texttt{while (run)\{} & 1 \texttt{while (run)\{} \\
2 \texttt{Acquire(fork1);} & 2 \texttt{Acquire(fork2);} \\
3 \texttt{if (TryAcquire(fork2)) break;} & 3 \texttt{if (TryAcquire(fork1)) break;} \\
4 \texttt{Release(fork1);} & 4 \texttt{Release(fork2);} \\
5 \texttt{\}} & 5 \texttt{\}} \\
6 \texttt{Release(fork1);} & 6 \texttt{Release(fork2);} \\
7 \texttt{Release(fork2)} & 7 \texttt{Release(fork1)} \\
\end{tabular}

The states are labeled according to the Threads state as a tuple. First part of the tuple is Thread1, second part is Thread2. State 1 corresponds to the state before line 1, state 2 corresponds to the state before line 3, state 3 corresponds to the state before line 4, state 4 corresponds to the state before line 6

Another problem can appear if the model checker is only applicable to terminating programs. Terminating programs have an acyclic state space, but realistically many concurrent systems have a cyclic state space which can be a big obstacle. Cycles resulting in a deadlock are often detected by model checkers. A livelock is very similar to a deadlock where the program is unable to make progress. In contrast to
a deadlock the program is not locked to on state, but rather locked in a cycle of states. This happens if the scheduler used is not fair to all processes and therefore gives one process more time to execute instructions and neglects the other. In Figure [1] we see such a livelock with this sequence of Operations: Thread1 performs **Acquire(fork1)** (1,1 → 2,1) and then Thread2 repeatedly performs **Acquire(fork2)** (2,1 → 2,2), **tryAcquire(fork1)** (2,2 → 2,3), **Release(fork2)** (2,3 → 2,1). A cycle is created so that Thread1 is unable to make progress. A method to tackle this problem is *Fair Stateless Model Checking* [11].

### 4 Iterative Context Bounding

Another technique to systematically and effectively explore the state space of a concurrent system is **Context Bounding**. Context Bounding is based on the typical depth-bounded search techniques, where the search may only explore a certain number of steps from the initial state. Iterative depth-bounded repeats the search with an increased bound if the previous search terminated. This technique is an under-approximation of the system and very useful for finding bugs and errors, however depth bounded search is only useful if those bugs appear within a small number of computation steps. That means if a depth-bounded search with a depth-bound d terminates, there are no errors in executions with at most d steps. Concurrent systems are very complex, which makes regular depth-bounded search inadequate, since the memory needed grows exponentially as the depth is increased. Nevertheless, it is proven that bugs related to concurrency can only appear after a number of context switches. Context switches are points in execution when the active process temporarily stops execution and another process starts or is resumed. This usually happens because of time slices (also called preemptions), timer-interrupts or system calls. In detail a context switch consists of the following activities in regard to processes on the CPU:

1. Suspending one process and storing its CPU state in the memory
2. Retrieving the context of the next process from memory and restoring it in the CPUs registers
3. Returning to the location of execution indicated by the program counter in order to resume the process

This sequence of activities is no problem for sequential programs, but in concurrent systems where multiple processes can share the same object in memory this can lead to unexpected behavior. We distinguish two kinds of context switches. There are **preempting context switches** (short preemptions) and **non-preempting context switches**. If a process voluntarily yields its execution, because it is blocking or terminates, we call it a **non-preempting context switch**. A **preemption** occurs when the scheduler forcefully suspends its execution, for example because of the expiration of a time slice. Concurrency errors can occur if a thread temporarily violates a global program invariant and then is suspended. Other threads that need this invariant will have an incorrect execution. In regard to concurrency only preempting context switches are interesting, which is why the Iterative context bounding algorithm only bounds the number of preemptions, but not the number of non-preempting context switches when exploring the state space. An unbounded number of non-preempting context switches means a program can be driven to termination without even causing a preemption as we can see in Figure [2]. For the program in Figure [2] there are exactly two execution orders without preemptions (abcd,c dab), while there are two execution orders with 2 preemptions (acdb,cadb) and two execution orders with 3 preemptions (acbd,cadb).

This behavior can be created by limiting the choices the scheduler can make, because a preemption can
Figure 2: Terminating program with and without preemption

**Shared Communication Variables:**
Int x,y;

**Thread1:**
1 x =1; (a)  
2 y =2; (b)  

**Thread2:**
1 y =3; (c)  
2 x =4; (d)  

blue: non-preempting, black: preempting context switch

only occur if the scheduler chooses a process, that is not currently running. Therefore, model checkers that use iterative context bounding are able to go deep in the state space and explore interesting behavior of the program, even with a low bound of preemptions. A search that runs out of resources after encountering \( c \) preemptions guarantees, that if there are any errors in the program, they require at least \( c+1 \) preemptions. This a valuable metric for the programmers, who are then able to approximate the complexity of the remaining bugs.

Before we look at the algorithm in detail, we look at some empirical information. A look at empirical values makes iterative context bounding a very attractive technique. In Figure 3 we see the percentage of reachable states the search has explored on the y-axis and the preemption bound on the x-axis. The program has executions with at least 35 preemptions, but we reach a state coverage of 90% with a bound of only 8. In Figure 4 we see a comparison to other state-space searches. The number of explored states is plotted against the y-axis on a logarithmic scale, while the x-axis shows the number of executions and thus represents the time needed. These two diagrams suggest that iterative context bounding is good at achieving high coverage within a low bound and low number of executions. In contrast to other, state-space searches, iterative context bounding has a significantly better coverage at a better rate. Even with a low bound of preemptions, that is significantly lower than the maximum possible number of preemptions, a high coverage can be achieved.

Now we take a closer look at the actual algorithm in Figure 5.
We begin our search at the initial state \( s_0 \) and explore executions with a certain preemption bound (\( currBound \)), which we increase after each iteration. This algorithm does not stop at a certain bound, but that can be easily implemented.
The main type of object the algorithm uses are **WorkItem**, which stores a state and a thread, as well as...
Figure 5: Iterative Context Bounding algorithm (from [12])

Input: initial state $s_0 \in State$

```
1 Object WorkItem { State state, TID tid };
2 Queue<WorkItem> workQueue;
3 Queue<WorkItem> nextWorkQueue;
4 
5 WorkItem w;
6 int currBound = 0;
7 for (t ∈ enabled (s_0) )
8 {
9   workQueue.Add( WorkItem(s_0, t));
10 }
11 
12 while (true) do
13 {
14   while (¬ workQueue.Empty() )
15     {
16       w = workQueue.Front();
17       workQueue.Pop();
18       Search(w);
19     }
20   if (nextWorkQueue.Empty() ) Exit();
21   currBound = currBound + 1;
22   workQueue = nextWorkQueue;
23   nextWorkQueue.Clear();
24 }
25 
26 
27 Search(WorkItem w)
28 {
29   WorkItem x;
30   State s;
31   s = w.state.Execute( w.tid );
32   if( w.tid ∈ enabled(s) )
33       {
34         x = WorkItem(s, w.tid );
35         Search(x);
36         for(t ∈ enabled(s) \ {w.tid} )
37           {
38             x = WorkItem(s, t);
39             nextWorkQueue.Push(x);
40         }
41     }
42   else {
43     for ( t ∈ enabled(s))
44       {
45         x = WorkItem(s, t);
46         Search(x);
47       }
48     }
49 }
```

two Queues workQueue and nextWorkQueue, which store WorkItem objects. A workItem w notifies the model checker to schedule Thread w.tid from state w.state. workQueue contains all the items that can be explored within the current bound. nextWorkQueue contains items whose execution would result in at least one additional preemption and thus exceeding the current bound.

First we initialize the workQueue in lines 8-11. For each enabled thread in $s_0$ we add a workItem to the queue.

Next we iteratively remove an item from the queue and invoke the procedure Search() on it (lines 15-20). After this has been done, it is guaranteed that we explored any executions with a maximum of currBound preemptions.

In the case that nextWorkQueue is empty, the algorithm terminates (lines 21-22). Otherwise we add all item from nextWorkQueue to workQueue, clear nextWorkQueue, and increase the preemption bound (lines 23-25).

The procedure Search() works recursively on an workItem w. First, this procedure executes thread $t = w.tid$ from state $s = w.state$ and reach a new state $s’$ (line 31). Since the algorithm is meant for concurrent systems, the thread may execute at most one visible operation (access a shared variable), after which we need to add a scheduling point.

In lines 32-35 we determine whether the same thread is enabled in $s’$ and schedule it for another step. If thread t was enabled in state s’ and we’d execute another enabled thread $t’$ from that state a context switch would occur. We do not want that and add a new workItem($s’, t’$ to nextWorkQueue for each enabled thread $t’$ in state $s’$ (lines 36-40).

If thread t is not enabled in state $s’$ a context switch cannot occur. Therefore, in lines 42-47 we execute Search() on each enabled thread $t’$ in state $s’$.

This algorithm does not cache states and therefore has the risk of not terminating on a cyclic state-space.
State caching however, can be trivially added. Depending on the model checker we use, we either get stateless or stateful model checking with this algorithm. CHESS is a stateless model checker, while ZING is a stateful model checker ([12]). A proof for this algorithm and further details can be found in Qadeer and Musavathi’s work [12].

5 Partial Order Reduction

Partial Order Reduction is an efficient method to reduce the size of the state space. Partial Order Reduction techniques exploit the fact that usually multiple sequences of transitions are equivalent, which means they lead to the same state. The name Partial Order Reduction stems from early versions that were based on the partial order model of program execution. A more fitting name would be model checking using representatives. As we said in Section 2, a state is defined by the transitions that lead to it starting from the root. If different sets of transitions lead to the same state, exploring all those branches would be a waste of time. Partial order reduction techniques group those equivalent branches into classes, so only one representative for each class needs to be explored. Partial Order Reduction techniques need a dependency relation definition between transitions to identify when distinct transitions are independent. A dependency relation that satisfies the criteria is a reflexive, symmetric binary relation on the transitions. The relation does not need to be transitive. Mazurkiewicz trace [10] formally defines under which conditions two traces are equivalent. The behavior we get from the reduced system is only a subset of the behaviors of the full state graph, but the behaviors that are not present do not add any new information. This kind of state space reduction is best suited for asynchronous systems, where concurrent transitions are interleaved. In synchronous systems transitions are executed simultaneously, which makes Partial Order Reduction inefficient.

The importance of reduction techniques such as Partial Order Reduction can be shown: Consider a concurrent system that consists of $n$ processes. Each process $p_i$ has an enabled transition $\alpha(a_i)$ in some state $s_i$ that leads to $s_i'$. Those concurrent transitions $\alpha(a_i)$ can be ordered in $n!$ different states. By applying partial order reduction, we group together all sets of transitions that lead to the same state. Then we only need to look at one particular ordering which represents its group. The number of states we need to consider is significantly reduced to $n+1$ instead of $2^n$.

Typically, there are several kinds of techniques to perform Partial Order Reduction. Some Partial Order Reduction algorithms work with persistent sets (variations are ample and stubborn sets). These are provably-sufficient subsets of the enabled transitions of state $s$. Enabled transitions that were not selected by the algorithm are guaranteed not to interfere with those that were selected. To generate those sets, the algorithms needs to exploit information about which branches are most probable. Usually information which transitions need to be considered is obtained from a static analysis of the program. Static Partial Order Reduction algorithms compute persistent sets immediately after reaching the state $s$. However, static analysis has limitations and persistent sets may be very large.

For example most static analyzer are unable to decide if transitions are independent or not in the following scenario:
Two enabled transitions $t1$ and $t2$ lead out of $s$ to access an array $a[]$ at different locations $n$ and $m$ ($a[n]$ and $a[m]$). Static analyzers may not detect if $n == m$. The result is that in many cases calculating persistent/stubborn sets is very imprecise, which means the state space cannot be pruned effectively.
Another main partial order reduction technique uses sleep sets [7]. This technique reduces the state space by extracting information on dependent enabled transitions in the current state and additionally uses information gained in the past of the search. This approach has originally been designed to detect deadlocks [6]. To detect those cycles in the state space, these techniques have to be combined with additional constraints. Similar to fair stateless model checking [11], transitions have to be selected fair in order to detect deadlocks and livelocks, as well as checking arbitrary safety and liveness. Since sleep set techniques only reduce the number of transitions, but not the number of states, if used alone they cannot avoid state space explosion.

Therefore sleep set techniques and persistent set techniques often are used simultaneously to complement each other.

In the next section 5.1 we discuss Dynamic Partial Order Reduction ([3],[16],[15]), which uses runtime information to generate much smaller persistent sets.

### 5.1 Dynamic Partial Order Reduction

*Dynamic Partial Order Reduction* is a technique that addresses the problems *Static Partial Order reduction* has. This technique uses runtime information to create much smaller persistent sets than a static analysis.

To achieve this, interactions between processes during a depth-first search on the execution tree are tracked. The information gained on dependency of transitions is then used to create *Backtrack sets* and dynamically update them during the execution of the program. These backtrack sets are sets of transitions that need to be explored, because they lead to branches that have not yet been explored and are not equivalent to a sequence of transitions that has already been explored. Those backtrack sets are a kind of persistent set. If all sequences of transitions, both in the normal persistent set and the backtrack set, have been explored and no new backtracking points have to be added the search terminates. The algorithm is guaranteed to find any deadlocks and bugs related to concurrency by then.

Dynamic Partial Order Reduction algorithms do not explicitly store previously visited states.

To describe the algorithm by Godefroid and Flanagan [3] in detail we need additional definitions

1. Let $S$ denote a sequence of transitions ($S \in T$)
2. $S_i$ is the transition $t_i \in S$
3. $S.t$ means we add transition $t$ to set $S$
4. $\text{dom}(S)$ returns the set $1, \ldots, n$
5. $\text{pre}(S,i)$ for $i \in \text{dom}(S)$ refers to state $s_i$
6. $\text{last}(S)$ is the state $s_{n+1}$ reached by transition $t_n$
7. $\text{proc}(S_i)$ refers to the process executing the transition $t_i \in S$
8. $\text{next}(s,p)$ is the next transition process $p$ executes from state $s$

A transition $t \in T$ is not limited to only one occurrence in a transition sequence $S$. If two transitions $t_1$ and $t_2$ are both enabled in the same state, we call those co-enabled.
Initially: Explore(Ø)

Explore(S) {
  s = last(S);
  for (Process p ∈ 𝒫) {
    if (∃ i == max({ j ∈ dom(S) | S_j is dependant and may be co-enabled with next(s, p) and i →_s p})) {
      E = {q ∈ enabled(pre(S,i)) | q == p or ∃ j ∈ dom(s) : j > i and q == proc(S_j) and j →_s p};
      if (E ≠ Ø) add any q ∈ E to backtrack(pre(S,i));
      else add all q ∈ enabled(pre(S,i)) to backtrack(pre(S,i));
    }
  }
  if (∃ p ∈ enabled(s) {
    backtrack(s) = {p};
    done = Ø
    while(∃ p ∈ (backtrack(s) \ done)){
      add p to done;
      Explore(S.next(s,p));
    }
  })
}

Additionally we need a partial order that helps us define under which circumstances two transition sequences belong to the same equivalence class:

The happens-before order, which is formally described as ”Mazurkiewicz’s trace” [10], is such a partial order, which →_s denotes. →_s for a transition sequence S=t_1,...,t_n is the smallest relation on {1,...,n} such that:

1. if i ≤ j and S_i is dependant with S_j then i →_s j
2. →_s is transitively closed

Note that only dependent transitions participate in such a relation. We can swap adjacent transitions t_i and t_{i+1} in a transition sequence S and get transition sequence S’ as long as t_i and t_{i+1} are independent. S and S’ belong to the same equivalence class. The formal definition of dependency of two transitions is beyond the scope of this work, but is addressed in [3].

To identify backtracking points we use a variation of the happens-before relation: i →_s p holds for i ∈ dom(S) and process p if either

1. proc(S_i) = p or
2. ∃k ∈ i+1,...,nsuchthat i →_s k and proc(S_k) = p

The method Explore(S) of the algorithm in [6] is called everytime our depth-first search on an execution tree reaches a new state s. S is the search stack that contains all the transitions we executed so far in our search. Initially (line 1), we execute Explore() with the empty stack as argument. In line 4 we set s to the state we are currently at, which is last(S).

After that we execute a loop for each process p ∈ 𝒫:
For line 6 we get the next transition next(s,p). Then we try to find a maximal i, so that the transitions S_i and next(s,p) are dependant, may be co-enabled and i →_s p. In other words we are looking for a transition S_i, so that next(s,p) and S_i are dependant (and co-enabled), while S_i may not have been executed by process p and no transition S_k (k ≥ i), executed by process p, exists that has a dependency relation with S_i.
If $S_i$ exists, $S_i$ and next(s,p) are dependent (e.g. a race condition). We need to add a backtracking point that refers to the state just before $S_i$ was executed (pre(S,i)). We then create a set E of processes (line 8). We add any process q from the set of enabled processes in pre(S,i) to E, such that q = p or there exists a transition executed by q that has a dependency relation with a transition that p executed.

If E is not empty (line 9) we can simply add each process from it to the backtrack point that refers to the state pre(S,i). In this set E are only processes we identified as necessary to enable next(s,p) from pre(S,i). Otherwise, if E is empty (line 10), it is necessary to add all enabled processes in pre(S,i).

When we’ve finished the loop and examined each process we continue with the next steps: We continue our search at state s. However, as long as there are processes in the backtracking set of state s, we need to execute those one by one (lines 16-18) until every one of those processes has been explored. Then we can call the state s “backtracked” and call Explore(S.next(s,p)).

This algorithm does not explicitly store previously visited states in memory, which makes it stateless. This is guaranteed, because we can perform backtracking by re-executing the program from it’s initial state or by using the checkpointing technique ([5]).

We are going to illustrate how this algorithm works with a small example.

**Shared Communication Variables:**

```plaintext
Int x, y;
```

**Thread1:**

1  y = 3;
2  x = 4;

**Thread2:**

1  x = 1;
2  x = 2;

For simplicity reasons we only consider two threads. We execute above code arbitrarily and get following exemplary execution order:

T2:  x = 3;  T2:  x = 2;  T1:  y = 3;  T1:  x = 4;

The algorithm will check whether there are dependent transitions with Thread1, before it executes the last transition Thread1: x = 4. We need to add a backtracking point before the last transition that has a dependency relation with Thread1 (and is not Thread1), which is transition Thread2: x = 2. As a result we need to explore following sequence:

T2:  x = 1;  T1:  y = 3;  T1:  x = 4;  T2:  x = 2;

Then the algorithm explores that second sequence and will add a backtracking point for Thread1 before T2: x = 1. As a result we need to explore:

T1:  y = 3;  T1:  x = 4;  T2:  x = 1;  T2:  x = 2;

The correctness of this algorithm is proven in [3].

As seen in Figure [7] Dynamic Partial Order Reduction is a significant improvement over Static Partial Order Reduction for a File System Benchmark example. Dynamic Partial Order Reduction is very efficient for up to 13 Threads, while Static analysis already struggles with only a few number of Threads.
Figure 7: Number of transitions explored for the File System Benchmark (from [3])

However, that may be only valid for that specific benchmark. Dynamic Partial Order Reduction might be less efficient for other programs and there might be instances where Static Partial Order Reduction performs better. Since Dynamic Partial Order Reduction arbitrarily picks the initial transition $t$ from each state and dynamically computes the persistent set with that transition $t$, Static Analysis may be able to compute a much smaller persistent set that doesn’t include $t$.

One problem Dynamic Partial Order Reduction has is cycles. Since it doesn’t store visited states re-exploration of the same state is difficult to avoid. A simple solution is to bound the depth of the search. Combining it with techniques that deal with cycles and liveliness would be optimal.

6 Partial Order Reduction for Context-Bounded State Exploration

Partial Order Reduction\textsuperscript{5} and Iterative Context Bounding\textsuperscript{4} are two key techniques for systematic testing of multithreaded programs. Each of those two techniques has its advantages. Partial Order Reduction techniques are able to efficiently reduce the complexity of the state-space search. On the other hand using Iterative Context Bounding we can scale our systematic exploration to large programs and can still go deep in the state space. It is empirically proven that most bugs related to concurrency can be found within a low bound of context switches. Additionally, iterative context bounding provides us with a valuable metric:

If a context-bounded search terminates with a bound of $c$ context switches, there may only be errors in the program with at least $c+1$ context switches.

**Partial-Order Reduction for Context-Bounded State Exploration**\textsuperscript{9} is a technique which combines those two different techniques. The standard case for a context bounded search is a *totally-ordered* execution. This means our context bound corresponds to the number of context switches in that totally-ordered execution. The context bound in *partially-ordered* executions however, is the minimum context-bound of all totally-ordered executions that correspond to this partially-ordered execution. If we applied context bounded restrictions to partial order reduction techniques, we would want to explore each partially-ordered execution with a maximum of $c$ context switches exactly once.

The naive approach prunes transitions if the partial order reduction algorithm requires it or if the transitions have more context switches than our context bound allows. This approach has some issues:
Figure 8: Unexplored State (from [9])

Shared Communication Variables:
Int x, y;

Thread1:
x = 1 ;
y = 1 ;
Thread2:
y = 2 ;

We illustrate the issue in figure 8. Our program has two threads, T1 and T2. We reduce our state space with partial order reduction, which uses sleep sets, and a context bound of 0. Our scheduler is not fair and thus T1 is executed before T2 in each state. Our algorithm prunes transition $2 \rightarrow 4$, because a context switch occurs. Transition $3 \rightarrow 4$ is pruned, because action $X:=1$ of T1 is an independent local action that is not relevant for concurrent interactions with action $y:=2$ of T2. Its not necessary to prune any of the other transitions, because no context switch occurs. The problem that is on hand here is that a sequence of transitions where the final value of y = 1 is not explored, even though the sequence $y:=2$, $x:=1$, $y:=1$ does not exceed our context bound. Therefore, a simple approach for combining partial order reduction with context bounding is not sufficient.

Partial Order Reduction for context bounded state exploration introduces an algorithm that solves this issue. This algorithm represents the state space of a program as a happens-before graph [8] and is able to store these graphs in a hashtable, so they can be accesses easily and efficiently. As we described earlier the algorithm needs to find context-bounds for each partial order, which is not an easy task, because it is NP-complete. The algorithm is able to solve the problem by using memorization. Partial Order Reduction for context bounded state exploration [9] has a more detailed description.

7 Related Work

Stateless Model Checking is a difficult topic. It was first proposed in Verisoft [4]. There are different approaches, but many approaches presuppose Sequential Consistency. The works [11] and [17] discuss this problem.

Partial Order Reduction:
There are a variety of Partial Order Reduction techniques [2] [6] [13] [14] . Some of them addresses a particular problem. For example [17] is concerned with the relaxed memory model. Dynamic Partial Order reduction is an improvement over static Partial Order Reduction techniques. First proposed in Flanagan and Godefroid’s work [3]. Further development led to TransDPOR [15]. There’s also an approach to apply Dynamic Partial Order Reduction to distributed systems [16].

Context Bounded Model Checking:
Iterative Context-Bounding was originally introduced by Qadeer and Musavathi [12]. Based on previous approaches of Context-Bounding they created a dynamic approach to the problem.
The combination of both partial order reduction and context bounding is a complex topic discussed in [9]. Another work [1] additionally addresses the weak memory model problem.

8 Conclusions, Results, Discussion

Stateless Model Checking

Stateless Model Checking is a viable alternative to Stateful Model Checking and especially to traditional stress testing. Stateless Model Checking addresses many of the issues the stateful counterpart has. The state explosion problem for larger, interleaved programs is such an issue. Explicitly storing and exploring all possible states of a program is almost impossible, since we only have limited memory and time to do so.

We have presented several approaches to Stateless Model Checking:

Partial Order Reduction:

Static Partial Order Reduction is only a small improvement over standard model checking techniques. There are stateful as well as stateless variants of it. In Figure 7 we can see that stateless Partial Order Reduction does not perform better than stateful Partial Order Reduction. Dynamic Partial Order Reduction 5.1 is however a significant improvement. It performs very well for a relatively high number of Threads. On top of that it does not need overly complex computations like static analysis needs. Coupled with the sleep set technique or using a combination of dynamic and static Partial Order Reduction, its efficiency can be improved even further. By nature it cannot deal with cycles very well, therefore we need to combine it with other techniques to use it to its full potential. (e.g. Fair Stateless Model Checking can do that)

Iterative Context Bounding

Iterative Context Bounding is another alternative to stateful model checking or systematic exploration of program behavior. By bounding only number of preemptions during search, we do not sacrifice the ability to go deep into the state space. We are able to scale it to large programs without any problems. Additionally, the context bound provides us a valuable metric.

Partial Order Reduction for Context-Bounded State Exploration

Partial Order Reduction for Context-Bounded State Exploration combines elements from both Iterative Context Bounding and Partial Order Reduction to create a very effective method of model checking.

All those stateless model checking techniques are not exclusive and can be combined to complement each other. Stateless model checking has high potential and is particularly viable for large concurrent programs.

9 Bibliography

References

REFERENCES


REFERENCES

